Beyond Code: Handling Variability in Art Artifacts in Mobile Game Product Lines

Vander Alves[1], Gustavo Santos[2], Fernando Calheiros[2], Vilmar Nepomuceno[2], Davi Pires[2], Alberto Costa Neto[1], Paulo Borba[1]

[1] Informatics Center - Federal University of Pernambuco, Brazil
{vra, acn, phmb}@cin.ufpe.br

[2]Meantime Mobile Creations, Brazil
{gustavo.santos, fernando.calheiros, vsn, davi.pires}@cesar.org.br

1. Introduction

In Software Product Line (SPL) engineering [1], while focusing on exploiting the commonality within the products, adequate support must be available for customizing the SPL core in order to derive a particular SPL instance. The more diverse the domain, the harder it is to accomplish this task. This, in some cases, may outweigh the cost of developing the SPL core itself. Therefore, variability management is at the core of SPL. Such variability spans various artifacts, from requirements to code and tests. Depending on the domain, additional artifacts should also be considered.

In particular, in Mobile Game Product Lines [2,3,4], art-related artifacts such as image and sound need to be addressed. Such artifacts are part of the core assets and their design and maintenance demand significant resources from organizations in this domain. Additionally, during product derivation, in which a game is ported to many devices, the great diversity of such devices complicates managing variability for sound and image. Failing to address variability in these artifacts adequately affect product derivation and also impact on other artifacts, such as code, thereby increasing the difficulty in managing its variation.

Based on our industrial experience, we address variability management of images and sound in the Mobile Game Product Lines. First, we briefly review variability issues in this domain and how they arise (Section 2). Next, we present some variability mechanism for such artifacts and reason how their choice is influenced by some factors, such as performance, binding time, and reusability (Sections 3 and 4). Finally, we evaluate how changing such mechanism affects variability management of code (Section 5).

2. Variability in Mobile Game SPL

Variability in Mobile Game SPL arises mostly due to a strong portability requirement and to great device diversity. Indeed, portability also becomes a central business issue in the contract between game developers and service carriers, since is it not economically viable for the latter to deploy a game for a few devices, thus representing a very small fraction of customers. Additionally, since device variability is great, this is especially relevant for games, which explore most device-specific optimized features to achieve competitive quality. Although these devices are organized by similarity into families by device manufactures, service carriers and developers, there still are dozens of families, and game developers must develop a game for most of them. This gives rise to SPLs with significant variability.

Based on our experience in this domain, we identified the most relevant device features and described the incurred variability. We have categorized variability in this domain. These categories are shown in Table 1:

Category	Description
Device specific variations	Differences regarding the device itself, like: • Screen sizes • Key codes • Sound playback approach • Presence of vibration API • Image transformation API
Known issues	General issues (bugs) encountered in more than one device that require a workaround
General variations	Support of multi-language and graphical font feature variations
Service carriers policies	Specific rules for deployment, like: • Network address of the server responsible for storing/retrieving information • Executable (JAR) file nomenclature
Feature variations	Presence or not of features like game ranking posting

Table 1. Variability in Mobile Game SPL

Addressing all these issues results in large SPLs. In fact, our SPLs currently have hundreds of instances.

3. Variability Mechanisms for Images

Image handling is a key activity in the game development process. Images are used for composing scenarios, characters, menus, and all the visual entities in a game. Considering the mobile device environment, the main factor causing image variations is the high number of device display sizes. Display size variation requires image resizing in such way that the figure elements fit into each display configuration. This way, besides code, images are product line assets affected by some factors that cause variations, thereby requiring corresponding variability mechanisms. In the following, we describe some of such mechanisms (automatic transformation, image decomposition, and location obliviousness) and reason on their choice by striking a balance among factors such as binding time, performance (space and time), and reusability.

The use of ***automatic transformations*** increases the reusability of images and demands a smaller effort from the graphics designer, who does not have to redraw all the images in a new scale for each device screen size. From the binding time perspective, there are two approaches to automatic image transformation: runtime and compile-time. In the former, the operations of image resizing, flipping, and color changing rely on an API and results in a decrease on the final executable size, which is the great advantage of this approach, since application size is one of the main development constraints for mobile devices [3]. This, however, has a moderate negative impact on performance, since the application now loads the image and transforms it, instead of just loading it; additionally, heap size usage also increases.

Compile-time automatic image transformation can be accomplished by the combination of image parameterization and image manipulation tools. In this approach, the game art is created for the largest screen size, and resizing parameters are set in a configuration file that is read by a tool creating resized images based on the reference image. It has the advantage of requiring less heap, and it does not have a negative impact on performance.

Both approaches of automatic image transformation may lead to visual quality loss, causing a bad game perception, making it impractical to use these operations. In such cases, the work of the designer is indispensable. The designer will have to create a new image for every transformation that cannot be accomplished using the aforementioned approaches, which leads to an increase in the size of the application's executable.

Image decomposition is a variability mechanism for decreasing the amount of images in the game and improving performance. It consists of dividing an image that is a part of an animation, or that can be reused by different elements of the game, into several parts, considering that some of these parts are repeated in more than one of the animation frames.

Two examples of usage of this technique are in Meantime's games mobile My Big Brother [5] and Ronaldinho Total Control [6]. In BBB5 ,there was only one image used for the torso of every character, and in Ronaldinho Total Control the main character was divided into several parts (arms, head, torso and legs), where the ones that moved were the arms, head and legs, so the torso image used in every frame of the animation was the same one. This required positioning the images to form the animation at runtime, as illustrated in Figure 1:

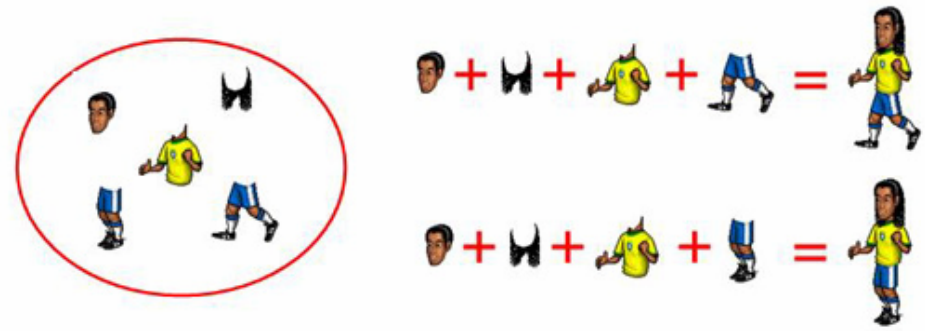

Figure 1. Image decomposition

Location obliviousness. Variability of device display sizes affects not only image sizes, but also implies in the variability of the specification of image items positioning within these images. For every screen size, there is a need for different constants specifying such positioning. This results in the need for many constants in the code, resulting in many magic literals. The use of well-known refactorings such as *Replace Magic Number with Symbolic Constant* [7] does not suffice to address this issue, since there may be a few hundreds of such literals, most of which can be of fine granularity (not only class constants, but also as local variables). Alternatively, macro usage may impact on the legibility and IDE integration, since the code does not compile with the macro symbols. The same happens with preprocessing, a frequently used technique to address this variability. Instead, we propose addressing this variability at runtime by *Location Obliviousness*.

Most of the games' images are created by the designers in the SVG format [8], which is a XML file describing the images' elements and their positions. Packaging a SVG file and parsing it to get the values needed to paint the images at the appropriate positions is not viable since SVG, being a XML file, is very verbose and, thus, has a large file size. In *Location obliviousness*, we convert the SVGs into a compact binary format, which has a small size and can be parsed efficiently at runtime. This format is called BVG (Binary Vector Graphics) and supports a subset of the elements defined in the SVG standard. The use of BVG effectively removes most of the image positioning code, making it easier to read and maintain, and allows the developer to focus on the game logic.

4. Variability Mechanisms for Sound

Sound is being used in more intelligent ways in mobile applications development, especially in games. It creates a different environment, making the game more involving. The diversity of devices, their resources and the need to keep a high quality sound may demand a great effort from the sound designer. Very often the designer has to create several sound artifacts to take the

maximum advantage of the devices' sound playback capabilities. As a consequence, each device family has different set of sounds.

Most devices work with MIDI audio files, but devices' constraints for sound playback lead to variations of sound artifacts that are managed by the creation of sound artifacts for more powerful devices and following a progressive reduction of audio channels, always trying to keep the quality and original sound identity. Indeed, this process cannot be completely automated. Removing some audio channel, voice or specific instrument from the audio object may cause a complete distortion of the original sound. This process is still quite dependent on the designer's artistic feeling.

Some porting tools [9] offer automatic transformations over audio resources, according to the target device, but the only guarantee is that the resulting transformed resource will be compatible with the corresponding device. However, it is frequently necessary to have a fine control over the resources file size, which requires a direct interference by the sound designer.

Sound variability can be managed by creating an audio core artifact, which is always reused as a key asset through the SPL, and the customizations are made via melody simplification and transformations between device specific formats.

Recently, another approach is being largely adopted by the mobile game industry: the use of a special MIDI format called Scalable Polyphony MIDI (SP-MIDI) [10]. In this standard, MIDI channels have a priority order and the sound designer decides which sound component goes to each priority level. This way, MIDI channels can be seen as SPL assets and core-assets are determined by higher priorities. Different devices with discrepant sound capabilities can use the same artifact, but each one use only a compatible amount of channels from it. The Figure 2 summarizes the overall sound production process.

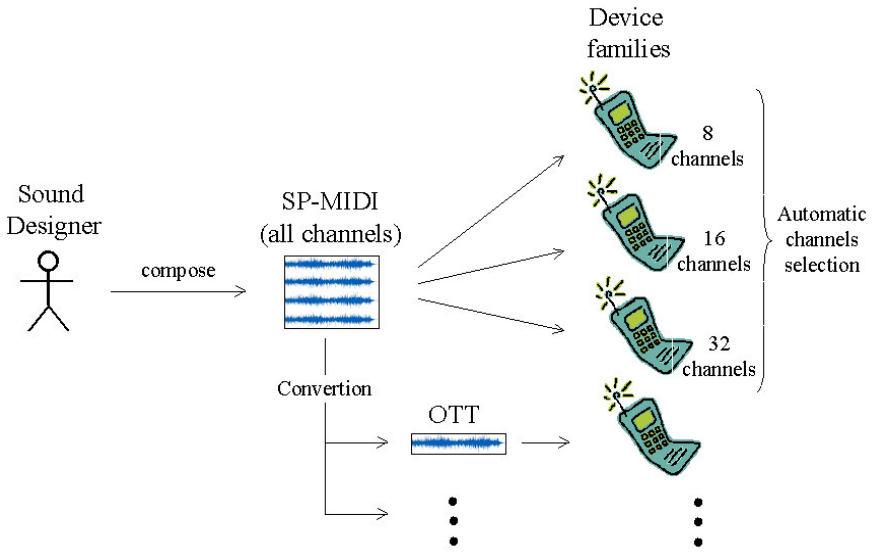

Figure 2. Sound as asset in Mobile Game Product Lines

Mobile devices are extremely restrictive regarding heap memory availability. As a direct consequence of this fact, the game programmer must be judicious about how much resources are being kept in the device heap memory at the same time. Two approaches can be applied to loading sounds: loading sounds at startup or on demand.

The first approach, ***loading sounds at startup***, consists in loading sound resources to memory during game startup process. As a consequence, it slows down game initialization and requires a greater amount of heap memory. On the other hand, it reduces the existing delay to load sounds during game execution and simplifies the codification.

Loading sounds on demand consists in allocating in memory sound resources when they are necessary and deallocating them as soon as possible. This approach reduces game initialization delay and also demands less heap memory during game execution. The downside is the increase in the game execution processing and the code complexity.

5. Variability Across Artifacts

The choice of the variability mechanisms for sound and images directly affects code variability. The API choice, resources allocation, execution mode of the artifacts generated by these mechanisms and how these artifacts are going to be represented and used inside the code influence the flow of execution and memory allocation, both heap and non-volatile.

The variability mechanism chosen for image representation can influence both the game's executable file and used heap memory. If the information about images positioning is not present in the loaded image object, it will have to be expressed as constants inside the code, thus increasing the executable size. Placing this information on text-based properties files to be read at runtime may degrade performance. The solution presented in Section 3 (Location Obliviousness) solves these problems. It decreases the number of code constants and, since it use a binary file, it occupies less space in the executable and is parsed more efficiently, demanding less processing power than a properties-based solution.

The choice of the variability mechanism for sound also affects the project's source code, as a consequence of different devices using different APIs and some of these APIs are more limited than others (such as the Nokia API for MIDP 1.0 devices). Additionally, in some cases the devices do not support sequential sound playback, making it necessary to create separate threads in the game flow so that playability is not affected. There are also restrictions on the type of the file supported by some devices. For these devices that contain that discrepancy it is necessary to use the file's content-type. The values that it may present are "audio/mid", "audio/x-mid", "audio/midi" and "audio/x-midi". Another variation is how the sound resources will be allocated: on demand, on devices that have low heap memory availability, or if they will be preloaded at the beginning of the game, which makes their execution response time faster. The creation of a uniform API for all devices that can be altered by code isolation using preprocessing directives is already the approach used in the industry, utilizing the preprocessor Antenna [11], a collection of Ant [12] tasks.

6. Summary

This position paper addresses on going work in exploring variability mechanisms for relevant artifacts in the domain of Mobile Game Software Product Lines. In particular, we show such mechanisms for images and sound and reason on the choice of such mechanism based on factors such as binding-time, performance, and reusability. Future work consists of refining a reasoning framework to encompass additional factors for the selection of such mechanism and exploring more closely the influence of such artifacts not only in code but also in tests.

References

1 P. Clements and L. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, 2002.

2 Vander Alves, Pedro Matos, Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and Evolving Mobile Games Product Lines. In Proceedings of the 9th International Software Product Line Conference (SPLC'05), volume 3714 of Lecture Notes in Computer Science, pages 70-81, September 2005. Springer-Verlag

3 Vander Alves, Ivan Cardim, Heitor Vital, Pedro Sampaio, Alexandre Damasceno, Paulo Borba, and Geber Ramalho. Comparative Analysis of Porting Strategies in J2ME Games. In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, pages 123-132, September 2005. IEEE Computer Society.

4 Vander Alves. Identifying Variations in Mobile Devices. Journal of Object Technology, 4(3):47-52, April 2005.

5 My Big Brother web site, http://www.meantime.com.br/games_meubbb.html, 2006.
6 Ronaldinho Total Control web site, http://www.ronaldinhomobile.com/, 2006.
7 M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: Improving the Design of Existing Code. Addison–Wesley, 1999.
8 World Wide Web Consortium, http://www.w3.org/Graphics/SVG/, 2006.
9 Unified Mobile Application frameworK web site, http://www.unifiedmobiles.com/, 2006.
10 SP-MIDI, http://www.midi.org/about-midi/abtspmidi.shtml, 2004.
11 Antenna web site, http://antenna.sourceforge.net/, 2006.
12 Ant web site, http://ant.apache.org/, 2006.